план 4-го занятия и примеры к нему
Sergey Melnik создан 7 месяцев назад

46a2f04b

planII_02.md 25,4 КБ

2.

##Основные вопросы

- Системы счисления - алгоритмы перевода и арифметические операции - с уклоном в двоичную систему.
- Машинное представление целых чисел.
  - Дополнительный код и его свойства. Переводы.
  - Целочисленные типы в Go: диапазоны; шестнадцатеричная (хексадецимальная) запись.

Основная идея

Числа живут в компьютере не только для того, чтобы их там хранить, но и чтобы исполнять с ними арифметические операции. С положительными числами и сложением в столбик более или менее понятно - видели на предыдущем занятии. А вот с отрицательными числами и вычитанием - пока непонятки. И вот мы в этот раз вводим дополнительный код благодаря которому сложение положительных и отрицательных чисел вообще никак не отличается. Соответственно, вычитание мы сводим к сложению. Заодно и умножение очевидно сводится к сложениям (и сдвигам, но на этом можно не заостряться). Если будет очень попутный ветер, то и деление можно рассмотреть, но при любых обстоятельствах хотя бы упомянем, что деление - это сравнения двух чисел, вычитания и сдвиги.

И таким образом мы вплотную подходим к построению сумматора, логическим операциям и формулам и логическим схемам.

Литература

```
Андреева,
        часть I, тема 4. Двоичное кодирование информации
                         тема 5. Двоичная арифметика
                         тема 6. Перевод чисел из двличной системы счисления в десятичную и обратно
                         тема 8. Компьютерное представление целых чисел
        часть III, глава 6. Целочисленная компьютерная арифметика
```

Разогрев. Небольшое повторение того, на чём закончили в прошлый раз.

Предыдущее занятие закончили на том, что в машине для хранения данных (пока только чисел) используются двоичные устройства - биты. Может быть быстренько повторить-поговорить о перимуществах именно такого хранения. И ещё по концовке предцдущего занятия вбросили тип uint8 - напомним. И сразу можем сказать, что у типа `uint8` есть другое название - `byte` (это именно алиас, а не другой, пусть и совместимый тип - их можно смешивать в одном выражении и присваивать друг другу). И на это фоне рассказали (или напомнили), что такое байт, может быть про другие, более крупные единицы: word или Кбайт/Мбайт/Гбайт. Но так, без фанатизма. Хотя можно потоптаться капельку на том, что 2^10^ и тысяча практически равны, хоть и не совсем, и к чему это приводит.

Немного поиграем для разогрева в переводы из двоичной системы в десятичную и обратно. Для перевода из десятичной в двоичную (и в любую другую) у нас был алгоритм "вычёрпывания", основанный на выделении максимальной степени числа P (P -основание системы счисления), которая входит в десятичное число. Можно показать алгоритм с делениями и записью остатков, который позволяет получать запись от младших разрядов к старшим. Доказательство корректности этого алгоритма, скорее всего, опустим. Показывать ли алгоритм - это как пойдёт.

Выполним два-три примера на сложение двух двоичных чисел, хорошо бы и какой-нибудь пример на умножение - умножаем в столбик, объясняя по ходу пьесы, почему этот алгоритм работает. В общем, в этой части делаем уклон на выполнение примеров-упражнений.

Машинное представление целых чисел.

Беззнаковые целочисленные типы

Двоичная система есть, всё готово. Забрасываем в детей все беззнаковые целочисленные типы:

```
uint8 и его алиас byte,  uint16, uint32, uint64
```

Определяем диапазоны и размеры каждого типа, сводим всё в примерно такую табличку

```
|имя типа | размер  |    диапазон       |   диапазон    |
|         | в битах |      точно        |    грубо      |
|---------|---------|-------------------|---------------|
|  uint8  |   8     | 0..2^8-1 = 255    |  до 255       |
|  uint16 |  16     | 0..2^16-1 = 65536 |  до 65 тыс    |
|  uint32 |  32     | 0..2^32-1         |  до 4 млрд    |
|  uint64 |  64     | 0..2^64-1         |  до 1.8*10^19 |
```

Особняком стоит тип uint. Не будем плодить терминологию зря, думаю, что говорить слова fundamental и generic типы ни к чему, но объяснить, что это такое, надо обязательно. Что вот это вот такой тип, зависящий от конкретной реализации, конкретной системы. Грубо говоря, для 32-битовых систем uint - это uint32, для 64-битовых - uint64. Неформально - это наиболее соответствующий конкретной системе программирования тип целых чисел без знака. Если вам нужна целочисленная беззнаковая переменная, то, если у вас нет особых причин выбирать именно тот или иной конкретный фундаментальный тип, то следует использовать uint.

Отрицательные числа. Дополнительный код и его свойства. Переводы.

А что с отрицательными числами? Понятно, что надо как-то хранить знак, например (и дети это предложат) потратить один бит на знак "+" или "-". И такой способ мог бы и прокатить, но оказывается есть что-то получше. И чтобы определиться, что здесь может быть получше, вспомним, что с числами надо бы выполнять арифметические операции. Ну, вот хотя бы сложение (которое лежит в основе и умножения, и, как мы сейчас увидим, всего остального)...

Давайте попробуем сформулировать, как складывать два знаковых числа. Знаковый здесь означает, что числа могут быть и положительными (плюс-числа), и отрицательными (минус-числа). И начинается... - если оба числа положительны, то мы их просто скаладываем, как положительные числа - если оба числа отрицательные, то знак у суммы будет минус, а величину суммы мы получаем, складывая величины обоих слагаемых - если знаки чисел различаются, то величину суммы мы получим, вычитая из большей из двух величинг меньшую, а знак суммы совпадает со знаком того слагаемого, величина которого больше

Ой, ужас. И это всё должен делать компьютер. Весь этот анализ, причём в самой базовой, самой глубинной операции. Я понимаю, что первые два варианта можно свести к одному, типа: если знаки совпадают, то величины складываем, а знак суммы совпадает со знаком обоих слагаемых. С другой стороны в описании пропущен случай (это я нарочно :)), когда складываем два числа с одинаковой величиной и разными знаками, т.е. когда сумма равна 0. И случай, когда одно или оба слагаемых равы 0. Я понимаю, что фигня вопрос, но ведь это всё должно учитываться всё равно. В общем, засада.

И тогда на сцену выступает весь в белом **дополнительны код**. Я для первого знакомства детей с допкодом всё-таки избегаю строго последовательно выводить допкод, как неизбежность. Это не очень сложно, это полезно, чтобы понять его естественность, но это, по-моему, сильный перегруз для первого знакомства. В принципе, довольно внятно и логично выводится допкод, причём на понятном десятичном примере, в книжке *Петцольд, глава 13*, но, повторю, я бы избегал. Моё предложение в данной ситуации в данных обстоятельствах: показать, что такое допкод, просто показать; привести исчерывающий набор примеров, как классно он работает для сложения (а, значит, и для вычитания) чисел со знаком, ну, и научиться находить машинное представление отрицателдьных чисел и наоборот - величину отрицательных чисел по их машинному представлению.

Как получается дополнительный код. Перевод отрицательных чисел в допкод и обратно.

Давайте рассмотрим однобайтовый беззнаковый тип uint8. Вот он весь:

```
        11111111        255
        11111110        254
        11111101        253
        11111100        252
        11111011        251
                . . .
        10000010        130
        10000001        129
        10000000        128
                                        <-  режем здесь
        01111111        127
        01111110        126
        01111101        125
        01111100        124
                . . .
        00000101          5
        00000100          4
        00000011          3
        00000010          2
        00000001          1
        00000000          0
```

А теперь возьмём этот список, разрежем его ровно посерединке - между 127 и 128 - на две части по 128 строк-чисел в каждой и верхнюю половинку перенесём параллельно вниз и приклеим к нижней:

```
 127    01111111        127
 126    01111110        126
 125    01111101        125
 124    01111100        124
```

```
 ...             . . .
   5    00000101          5
   4    00000100          4
   3    00000011          3
   2    00000010          2
   1    00000001          1
   0    00000000          0
                                              <-  склеиваем здесь
  -1    11111111        255
  -2    11111110        254
  -3    11111101        253
  -4    11111100        252
  -5    11111011        251
                       . . .
-126    10000010        130
-127    10000001        129
-128    10000000        128
```

В левой колонке - десятичные числа, соответствующие машинной записи. Вот это и есть допкод. Почему дополнительный? Осмелюсь предположить, что это потому, что мы записывем отрицательные числа как дополнение до 100000000 (тут 8 нулей, в общем случае M нулей, M - разрядность числа), т.е. сколько записи не хватает до 100000000. Например, машинная запись числа -14 - это двоичное число, которому недостаёт 1110 (т.е. 14) до 100000000. Следовательно, чтобы получить машинную запись числа -14 надо из 100000000 вычесть 1110. Проделаем это упражнение и получим 11110010.

Вычитать из 100000000 - геморрой. А попробуем вычитать не из 100000000, а из числа 11111111 (8 единиц), которое на 1 меньше, чем 100000000. О, вычитать из 1111111 очень легко и приятно - надо просто заменить в том числе, которое вычитаем все 0 на 1, а 1 на 0. Конечно, сначала надо дополнить это число нуями до 8 битов. 14 = 1110 = 00001110. 11111111 - 00001110 = 11110001. Остаётся прибавить к результату 1, чтобы компенсировать переход от 100000000 к 11111111. 11110001 + 1 = 11110010. Разумеется, получили тот же ответ.

В качестве упражнения возьмём ещё какое-нибудь не очень большое отрицательное число и нйадём его машинную запись. Например -25:

```
25 переводим в двоичную -> 11001,
дополняем до 8 битов -> 00011001,
инвертируем биты (заменяем 0 на 1, 1 на 0) -> 11100110,
прибавляем 1 -> 11100111.
```

Конечно, в случае 16-, 32-, 64-битовых величин хранение и способ перевода точно такие же.

Знаковые целочисленные типы

И теперь, в дополнение к первой табличке, или, скорее, расширяя её, получаем все типы для хранения целых чисел со знаком:

```
int8,  int16, int32, int64
```

Определяем диапазоны и размеры каждого типа, сводим всё в подобную табличку:

```
| имя типа| размер  |    диапазон   |   диапазон          |
|         | в битах |      точно    |    грубо            |
|---------|---------|---------------|---------------------|
|  int8   |   8     | -2^8 ..2^8-1  | -128..127           |
|  int16  |  16     | -2^15..2^15-1 | -32000..32000       |
|  int32  |  32     | -2^31..2^31-1 | -2 млрд .. 2 млрд   |
|  int64  |  64     | -2^63..2^63-1 | -9*10^18 .. 9*10^18 |
```

И опять особо про тип int. Главное правило: если у вас нет особых причин выбирать именно тот или иной конкретный фундаментальный тип для хранения целого числа, которое может быть и положительным и отрицательным, то следует использовать int.

Сложение чисел, записанных в допкоде.

У нас есть машинные записи

```
-14 : 11110010
-25 : 11100111
 14 : 00001110
 25 : 00011001
```

- Сложим тупо в столбик машинные записи 14 и 25, получаем 00100111. И это действительно 39. Пока ничего особенного нет.

- Сложим теперь, также тупо в столбик, машинные записи 14 и -25, получаем 11110101. Получили запись отрицательного числа - старший бит 1. Что это за число? Проделываем все операции по переводу числа в машинную запись в обратном порядке: вычитаем 1 -> 11110100, инвертируем биты -> 00001011, переводим в десятичную -> 11, меняем знак -> -11. О, получилось. Странно, но получилось.

Рассмотрим другие случаи.

- -14 + 25. Складывая в столбик машинные записи - 11110010 и 00011001 -, получаем 100001011 - девять двоичных цифр. Выкинем старший бит (как в счетчике, который после 99999 переходит в 00000), остаётся 00001011, а это именно 11, которые мы ждали.

- -14 + (-25). Складывая в столбик машинные записи - 11110010 и 11100111, получаем 111011001 - опять девять двоичных цифр. Опять выкидываем старший бит, остаётся 11011001 - запись отрицательного числа. Переведём машинную запись в десятичную: 11011001 минус 1 -> 11011000, инвертируем биты -> 00100111, переводим в десятичную -> 39, меняем знак -> -39.

- Можно ещё для полноты картины сложить 14 и (-14) или 25 и (-25). И в том, и в другом случае получим после сложения в столбик 100000000. А как же, мы же именно так и строили код отрицательного числа - это сколько не хватает до 100000000. Отбрасывает лишний старший бит и получаем 0.

Вау! И это реально вау! **Складывая числа, записанные в допкоде, мы вообще игнорируем знак числа, мы про него даже не знаем.** Ну, т.е. мы-то знаем, а вот комппьютер, точнее, его суммирующее устройство - не знает. Мы подслвываем ему две машинных записи, например 11100111 и 00001110, он их складывает и выдаёт 11110101. А дальше уже вопрос трактовки результата. Если мы думаем, что мы давали сумматору записи чисел 14 и -25, то трактуем ответ как доп код и получаем число -1. Если же мы думаем, что это были беззнаковые записи, т.е. складывади числа 14 и 231, то трактуем результат как беззнаковую запись и ответ получается 245, вполне ожидаемо.

И таким образом мы свели вычитание к сложению, а сложение знаковых чисел перестало быть таким ужасным с кучей случаев - случаев теперь вообще нет. Ура допкоду!

Шестнадцатеричная запись. Шестнадцатеричные цифры.

Вот тут у меня есть сомнения. Можно по-честному ввести понятие 16-ричной системы, показать, что она хорошая и т.д. Но ведь мы довольно редко используем шестнадцатеричную запись непосредственно - мы, как правило, используем 16-ричную запись для того, чтобы быстро-быстро получить битовое представление, расщепляя 16-ричную цифру на 4 двоичных, либо наоборот. И я бы именно так и говорил: 16-ричная запись получается путём группировки полубайтов в одну 16-ричную цифру. А чтобы делать это быстро-быстро, напишем и выучим таблицу:

```
| 0 |  0 |  0000 |
| 1 |  1 |  0001 |
| 2 |  2 |  0010 |
| 3 |  4 |  0011 |
| 4 |  5 |  0100 |
| 5 |  6 |  0101 |
| 6 |  7 |  0110 |
| 7 |  8 |  0111 |
| 8 |  9 |  1000 |
| 9 | 10 |  1001 |
| A | 11 |  1010 |
| B | 12 |  1011 |
| C | 13 |  1100 |
| D | 14 |  1101 |
| E | 15 |  1110 |
| F | 16 |  1111 |
```

Возможно есть смысл дать детям эту табличку на бумаге. И поиграемся немножко в переводах 16-ричная запись <--> двоичная запись.

fmt.Printf - %b, %x, %X, fmt.Sprintf

Я закидываю пару примеров на двоичный/шестнадцатеричный вывод. Заметим, что просто выводятся числа в соответствующей записи, а не машинное представление - лидирующие нули не выводятся, а отрицательные числа выводятся со знаком -. Примеры `II_01a.go`, `II_01b.go`

Практика

Конечно, упражнение на перевод числа в машинную запись и обратно выглядит очень естественным и уместным. И тут некое повторение устройства строки, что приближает нас к представлению рун, к UTF8. Но, на мой взгляд, очень важно дать в этот раз упражнение на сложение двух двоичных чисел, представленных строкой из 0 и 1, и результат, конечно, представляется так же. Даже если не впишется дать это задание, чтоб делали на практике, то очень жедательно его на практике хотя бы подробно обсудить, чтобы делали дома. Например, подсказать детям про выравнивание длин за счет дописывания лидирующих нулей в короткое слагаемое - сильно упрощает организацию цикла; очень хорошо бы, чтобы прозвучали слова "перенос" или "бит переноса" вместо слов "и один запоминаем" - разница в том, что перенос может быть 0, существенно стандартизуя процесс. И хорошо бы обойтись вовсе без оператора "+", сделать всё на if'ах. В общем, по возможности близко подвести-разогреть детей к схеме сумматора. Конечно, возможны и другие упражнения.

Упражнения на практику: - Вводится строка. Проверить, является ли она 16-ричной записью натурального числа. - Перевод из двоичной записи в шестнадцатеричную и обратно. Ввод и вывод - строки. - Вводится число знакового типа. Найти его машинную запись. Выводить результат как a. двоичную строку b. 16-ричную строку
- Длинная арифметика с двоичными числами: числа задаются как строки из 0 и 1. Вычислять a. сумму чисел; b. произведение чисел - моделируя умножение в столбик и базируясь на уже имеющемся сложении.